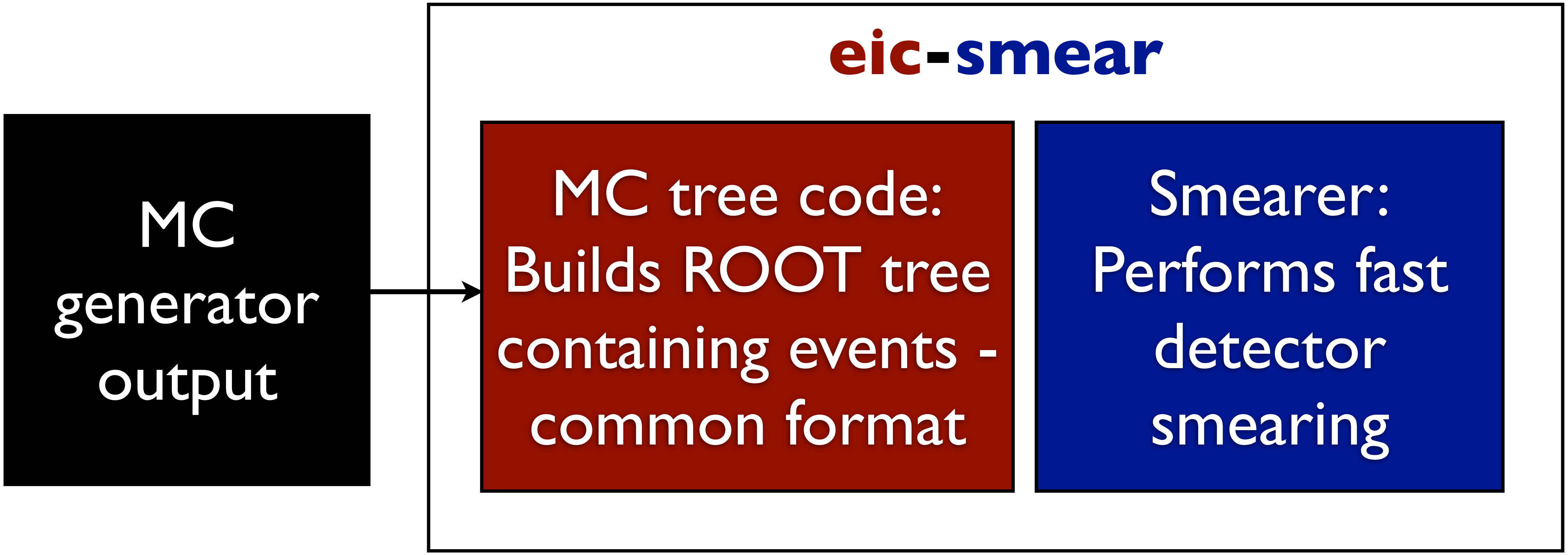
eic-smear overview

EIC task force meeting

Thomas Burton

14th August 2014

- What **is** it?
- ... and what is it **not**?
- How to **use** it
- **Ask me anything - ‘cause it’s your last chance :D**

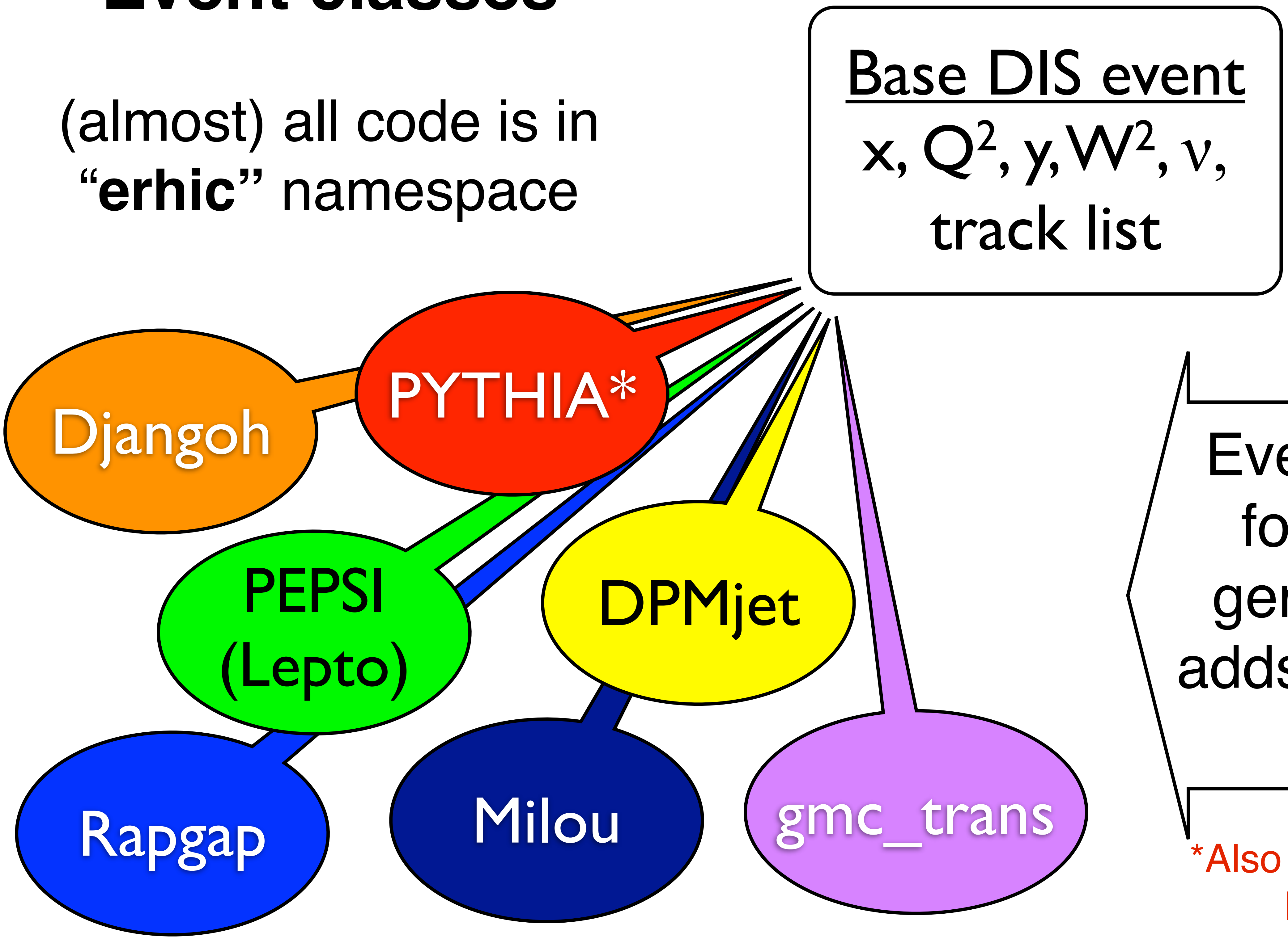

eic-smear
MC generator output
MC tree code: Builds ROOT tree containing events - common format
Smearer: Performs fast detector smearing

- **C++** code using **ROOT**
- Builds with **configure/Make** or **CMake**
- Stable version 1.0.3 works on Linux, OS X 10.6+
- Single **`libeicsmear.so`**
- Think of it as a “tool kit” rather than a standalone programme

MC tree code

Event classes

(almost) all code is in
“**erhic**” namespace



Base DIS event
x, Q^2, y, W^2, ν,
track list

defines
common
format

Event class
for each
generator
adds specific
data

*Also some p+p support,
but only basic

Standard format: macros can analyse different generators **without change**

Common ASCII format

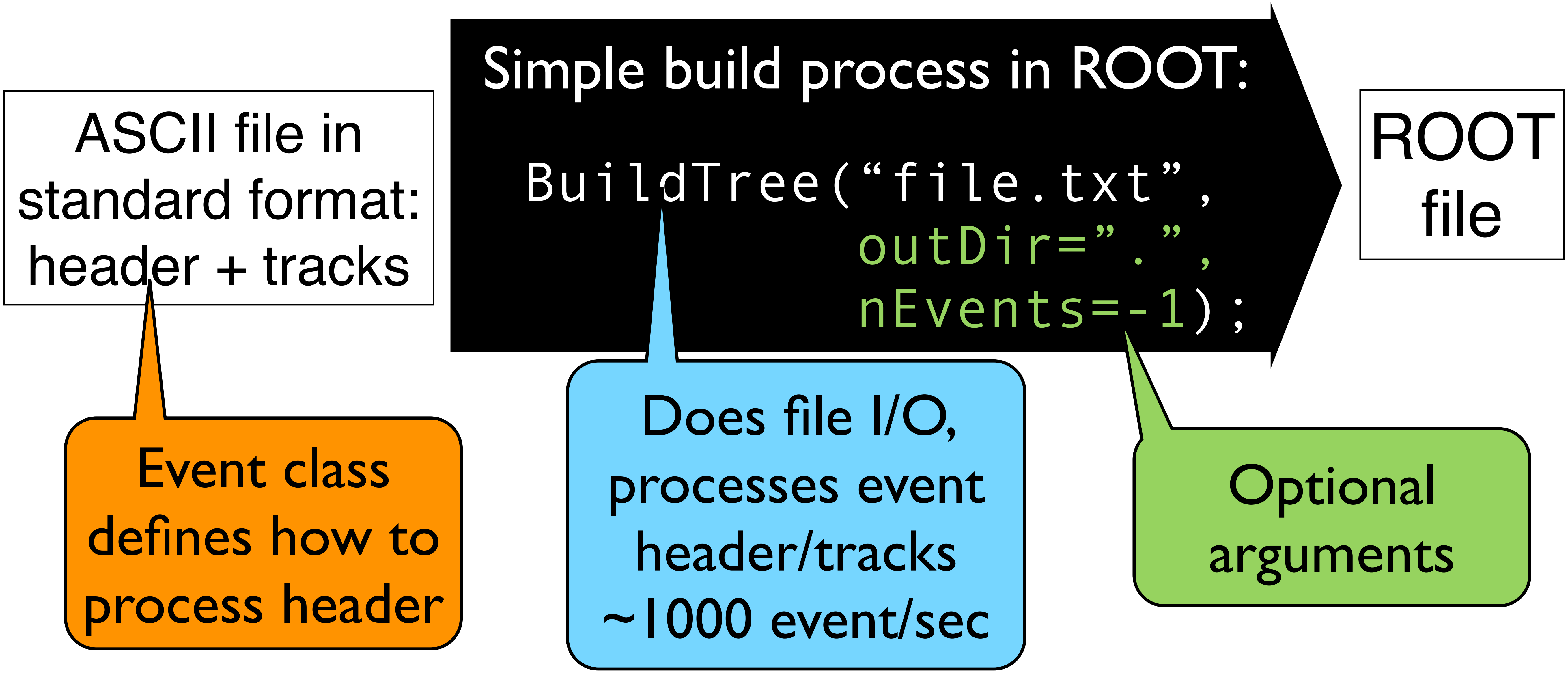
```
<generator name> EVENT FILE
============================================
<generator-specific event variable names>
============================================
Track variable names
============================================
0 <generator-specific event data>
============================================
1 KS KF parent child1 childN px py pz E m x y z
2 KS KF parent child1 childN px py pz E m x y z
...
N KS KF parent child1 childN px py pz E m x y z
=============== Event finished ===============
... <repeat event structure>
```

6-line file header

$(N_{tracks}$+3)-line event

https://wiki.bnl.gov/eic/index.php/PYTHIA

Building a tree from ASCII file



ASCII file in standard format: header + tracks

Simple build process in ROOT:

```
BuildTree(“file.txt”,
                outDir=”.”,
                nEvents=-1);
```

ROOT file

Event class defines how to process header

Does file I/O, processes event header/tracks ~1000 event/sec

Optional arguments

Yields “file.root” containing a TTree called “EICTree”

This is all the end-user has to run

(PYTHIA can also support direct ROOT output: http://svn.racf.bnl.gov/svn/eic/Generators/pythiaeRHIC/)

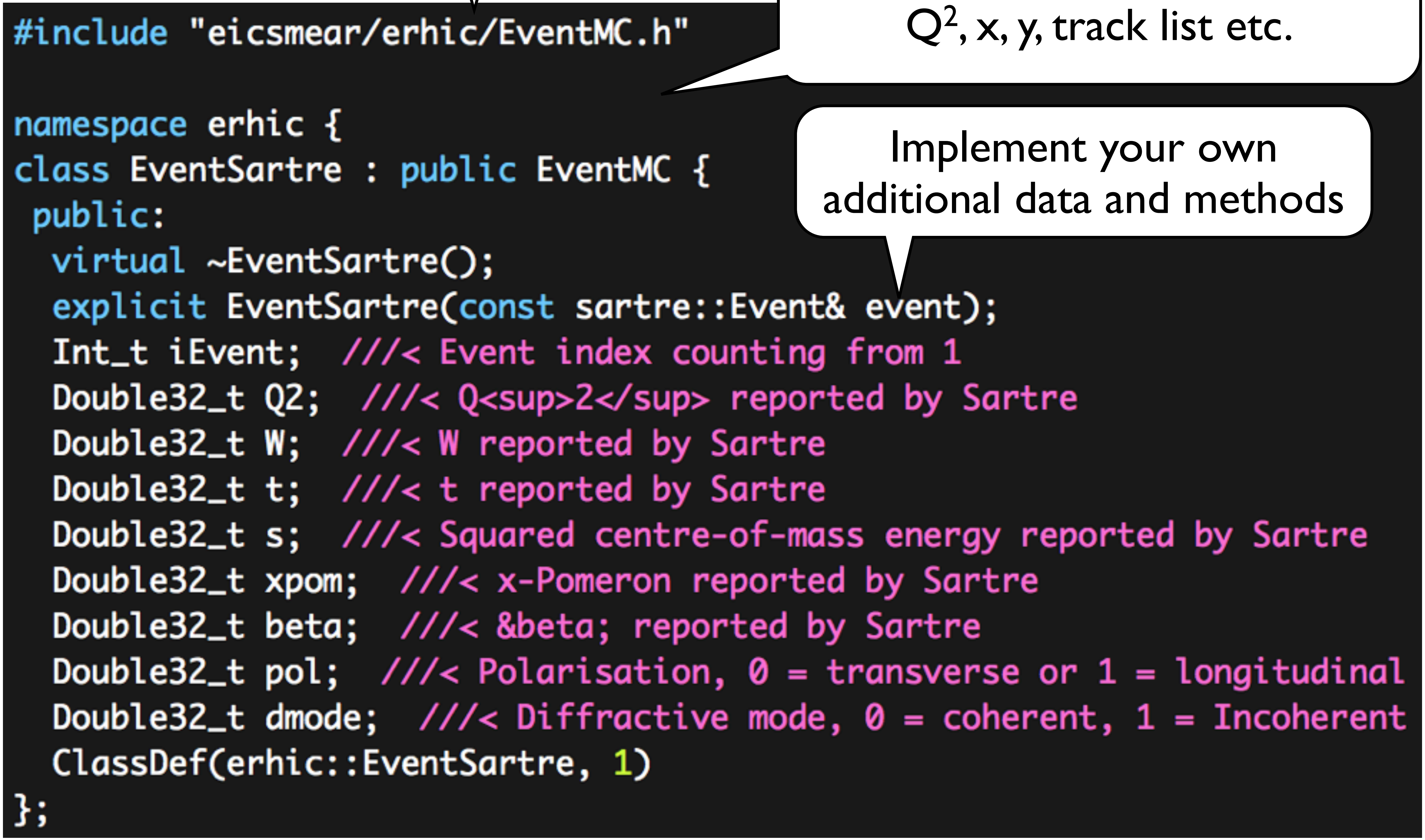
Extensibiliy - adding an event

- What if you have a new generator?
  - Not supported natively
  - Maybe has its own output format
- Can define your own event class, inheriting from an eic-smear class
  - e.g. what if we want to add **Sartre** support?

Include eic-smear event header

Inherit your class from it. Provides basic DIS functionality - Q^2, x, y, track list etc.

Implement your own additional data and methods



```
#include "eicsmear/erhic/EventMC.h"

namespace erhic {
class EventSartre : public EventMC {
 public:
  virtual ~EventSartre();
  explicit EventSartre(const sartre::Event& event);
  Int_t iEvent;  ///< Event index counting from 1
  Double32_t Q2;  ///< Q<sup>2</sup> reported by Sartre
  Double32_t W;  ///< W reported by Sartre
  Double32_t t;  ///< t reported by Sartre
  Double32_t s;  ///< Squared centre-of-mass energy reported by Sartre
  Double32_t xpom;  ///< x-Pomeron reported by Sartre
  Double32_t beta;  ///< &beta; reported by Sartre
  Double32_t pol;  ///< Polarisation, 0 = transverse or 1 = longitudinal
  Double32_t dmode;  ///< Diffractive mode, 0 = coherent, 1 = Incoherent
  ClassDef(erhic::EventSartre, 1)
};
```

Smearing

What is(n't) it?

- Utility for smearing of MC output
- It's **NOT** a replacement for Geant!
- But, if you are asking...

"Given a (known) detector performance, how well can I measure some physics observable(s)?"

or

"If I need to measure X with to some precision, what detector performance do I need?"

... then maybe it is for you

Architecture

- Originally written by Michael Savastio (student)
- Fast - thousands of events/second
- Smears
  - tracks: p, E, angle, ID
  - DIS kinematics: x, Q2, y

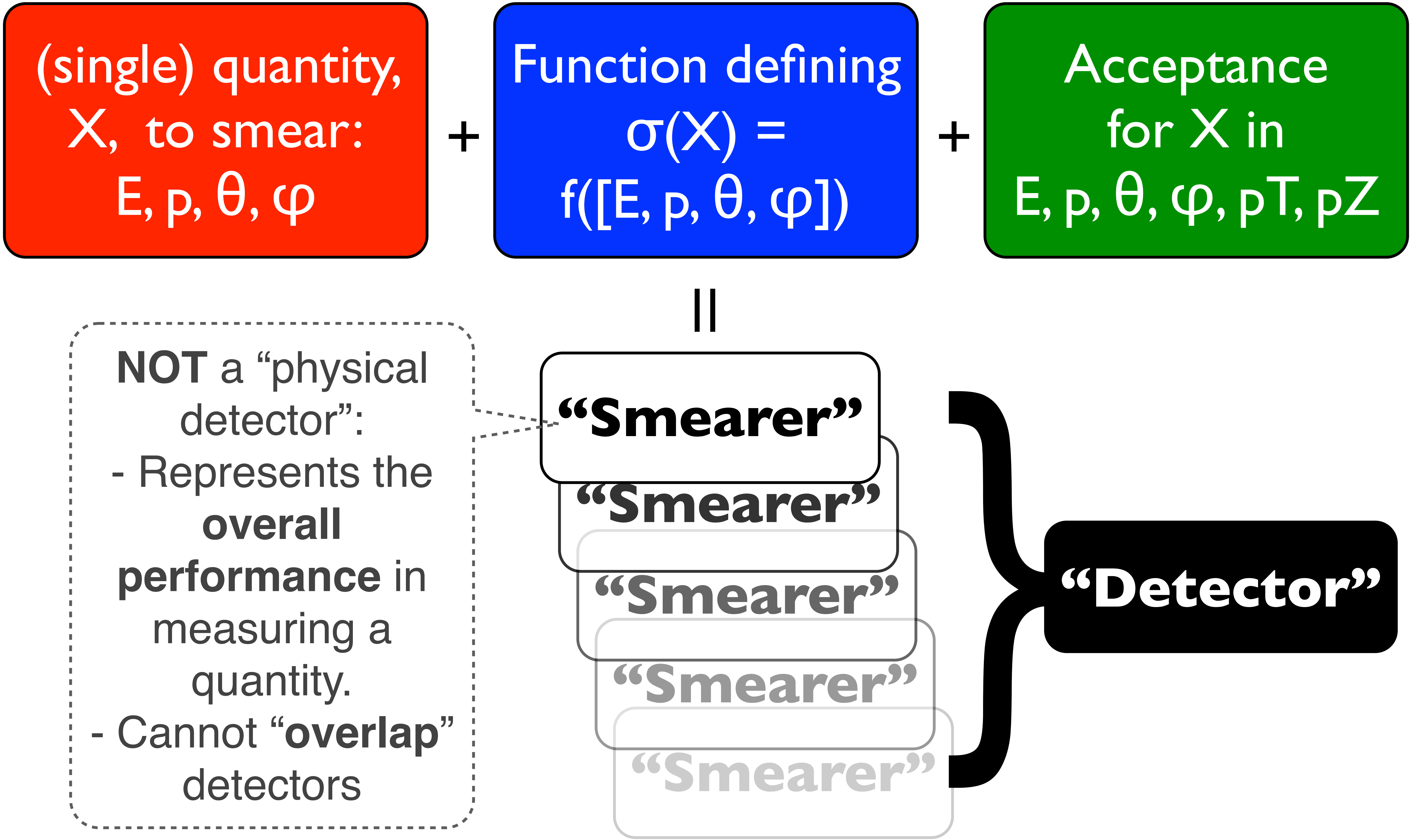
Output "ParticleMCS" track is stripped-down version of normal MC track

- **Not** specific to any generator
  - Same smearing specification works for **all** generator output that follows the common format

(almost) all code is in
"**Smear**" namespace

The idea

Has no “default behaviour”: you must define everything



(single) quantity, X, to smear: E, p, θ, φ

+

Function defining σ(X) = f([E, p, θ, φ])

+

Acceptance for X in E, p, θ, φ, pT, pZ

||

“Smearer”

“Smearer”

“Smearer”

“Smearer”

“Smearer”

“Detector”

NOT a “physical detector”:

- Represents the **overall performance** in measuring a quantity.
- Cannot “**overlap**” detectors

How to use it

- Write a ROOT script:

Simple Devices define σ(X) via text string

```
Smear::Detector createDetector() {
  // Resolution in momentum, sigma(P).
  // sigma(P) = 0.4%P + 0.3%P^2.
  Smear::Device tracking(“P”, “0.004 * P + 0.003 * pow(P, 2)”);
  // Resolution in energy, sigma(E) = 14% * sqrt(E)
  // 3rd argument == 1 -> smear only photons & electrons.
  Smear::Device emcal("E", "0.14 * sqrt(E)", 1);
  // Add devices to a Detector.
  Smear::Detector detector;
  detector.AddDevice(tracking);
  detector.AddDevice(emcal);
  return detector;
}
```

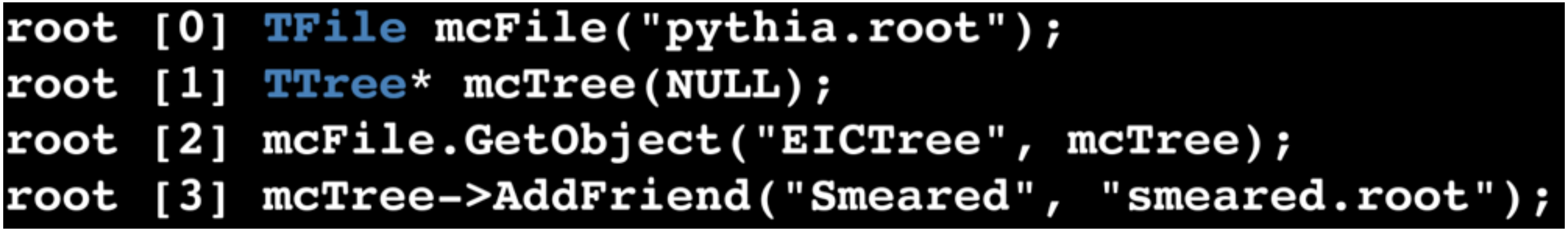
- Smear your ROOT tree:

Handles event loop, file I/O

```
root[0] SmearTree(createDetector(), “mc.root”, “smeared.root”);
```

Output

- Gives a new tree in common MC event format
  - scripts for MC events work on smeared events
  - Tree just named “Smeared”
- Easy to analyse with TTree “friend” mechanism:



```
root [0] TFile mcFile("pythia.root");
root [1] TTree* mcTree(NULL);
root [2] mcFile.GetObject("EICTree", mcTree);
root [3] mcTree->AddFriend("Smeared", "smeared.root");
```

- Only operates on **final-state** particles...

Output

	MC	*Smeared*
Event 1	Q2, x, y	*Q2, x, y*
	Particle 1	*NULL*
	Particle 2	*NULL*
	Particle 3	*Particle 3*
	...	...
	Particle N	*Particle N*
Event 2	Q2, x, y	*Q2, x, y*
	Particle 1	*NULL*
	...	...

If a particle is

1.not final-state* OR

2.not in the detector

➔ store NULL pointer otherwise store particle and smear its properties

- Keeps 1-to-1 matching between tracks in MC and smeared trees

*exception: initial beam particles **are** copied

Acceptance

- Each **Smearer** has an associated **Acceptance**
  - Acceptance is made of one or more “**Zones**”
  - Each Zone defines (p, E, theta, phi, ...) region
  - Zones needn’t be contiguous
  - Particles are only accepted if they match at least one Zone
- By default accepts everything
- Can also define other acceptance criteria
  - “**Genre**” - hadronic, electromagnetic
  - **Charge** - neutral, charged

Output - important note

- Different quantities may therefore have different acceptance e.g. smear
  - E for $-4 < \eta < 4$
  - p for $-3 < \eta < 3$
- Only smears quantities for which particle is in acceptance
- store zeros for quantities if particle outside acceptance
  - e.g. above, for particle at $\eta = 3.5$
  - E will be smeared
  - p will store zero

Other “Smearers”

- Extensible to more specialised devices
  - “Bremmstrahlung” class mimics electron energy loss by photon emission
  - Particle ID classes allow definition of a particle-misidentification matrix e.g. HERMES RICH

Other “Smearers”

- Generic “tracker” class, implementing

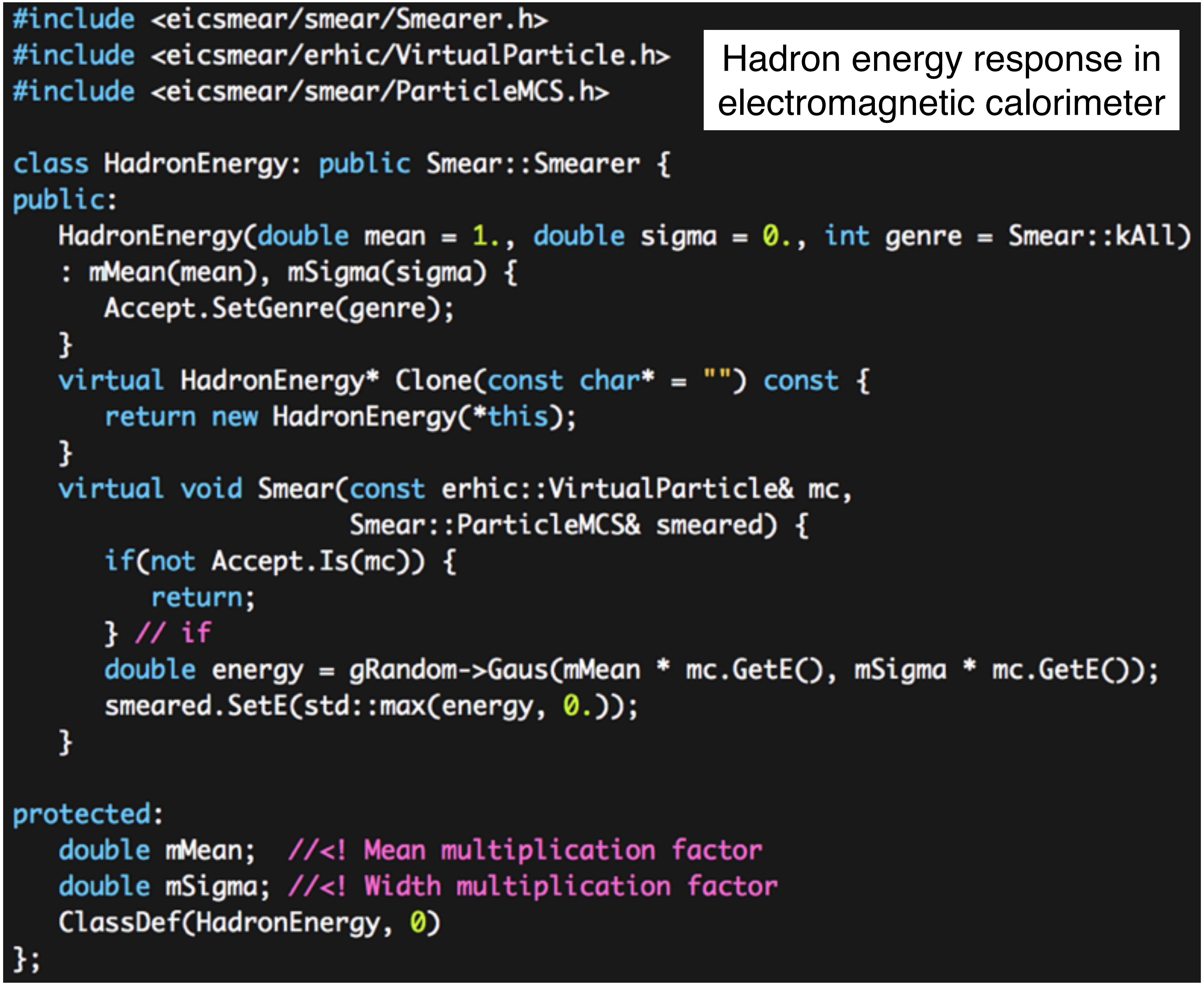
intrinsic resolution

$$\left|\frac{dp}{p}\right|_i = \frac{p}{0.3B_T}\frac{\sigma_{r\phi}}{(L')^2}\sqrt{\frac{720}{n+4}}$$

multiple scattering

$$\left|\frac{dp}{p}\right|_{MS} = \frac{1}{0.3B_T}\frac{0.0136}{L\beta\cos^2(\gamma)}\sqrt{n_{rl}}$$

Extensibility - define a smearer



Hadron energy response in electromagnetic calorimeter

```
#include <eicsmear/smear/Smearer.h>
#include <eicsmear/erhic/VirtualParticle.h>
#include <eicsmear/smear/ParticleMCS.h>

class HadronEnergy: public Smear::Smearer {
public:
   HadronEnergy(double mean = 1., double sigma = 0., int genre = Smear::kAll)
   : mMean(mean), mSigma(sigma) {
      Accept.SetGenre(genre);
   }
   virtual HadronEnergy* Clone(const char* = "") const {
      return new HadronEnergy(*this);
   }
   virtual void Smear(const erhic::VirtualParticle& mc,
                      Smear::ParticleMCS& smeared) {
      if(not Accept.Is(mc)) {
         return;
      } // if
      double energy = gRandom->Gaus(mMean * mc.GetE(), mSigma * mc.GetE());
      smeared.SetE(std::max(energy, 0.));
   }

protected:
   double mMean;  //<! Mean multiplication factor
   double mSigma; //<! Width multiplication factor
   ClassDef(HadronEnergy, 0)
};
```

Access and documentation

Read about it:

https://wiki.bnl.gov/eic/index.php/Eic-smear + links

Get it:

Just run directly from EIC nodes OR

Download tarballs from the above page + follow build instructions OR

svn checkout http://svn.racf.bnl.gov/svn/eic/Utilities/eic-smear/trunk eic-smear

Run it:

```
root[0] gSystem->Load(“/path/to/libeicsmear”);
```

This is done automatically if you run the EIC logon scripts:

https://wiki.bnl.gov/eic/index.php/Computing